6 Advanced data structures

6.1 Fibonacci heap

The Fibonacci heap is a fast implementation of the priority queue.

Abstract data type PriorityQueue

```
ADT PriorityQueue:
  # Holds a dynamic collection of items.
  # Each item has a value/payload v, and a key/priority k.

  # Extract the item with the smallest key
  Pair<Key,Value> popmin()

  # Add v to the queue, and give it key k
  push(Value v, Key k)

  # For a value already in the queue, give it a new (lower) key
  decreasekey(Value v, Key newk)

  # Sometimes we also include methods for:
  # merge two priority queues
  # delete a value
  # peek at the item with smallest key, without removing it
```

Dijkstra's algorithm uses a priority queue to keep track of vertices to visit. On a graph with V vertices and E edges, Dijkstra's algorithm might

- make V calls to `popmin()`
- make E calls to `push()` and/or `decreasekey()`

Since E can be $O(V^2)$, we want an implementation of PriorityQueue with very fast `push()` and `decreasekey()`. The Fibonacci Heap was developed by Fredman and Tarjan in 1984, specifically to speed up Dijkstra's algorithm.

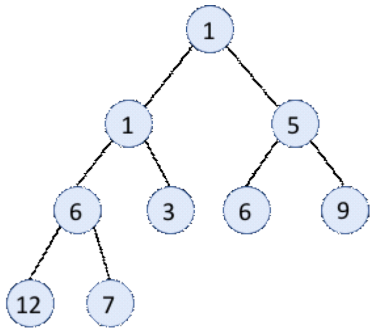
(The code for Dijkstra's algorithm in Section 5.5 also needs to test whether a value is already in the PriorityQueue. We'll assume that this piece of information is stored with each value, so it can be accessed in $O(1)$ time. We won't include this test in the ADT.)

Other implementations of PriorityQueue

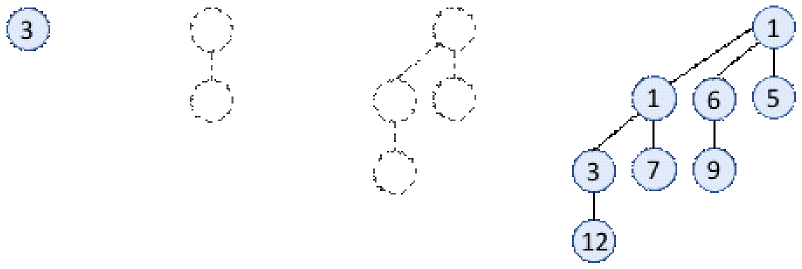
We've seen two other implementations of PriorityQueue so far: the binary heap, and the binomial heap. If there are n items in the PriorityQueue, these implementations have the following running times.

	binary heap	**binomial heap**	**Fibonacci heap***
peekmin	O(1)	O(log n)	O(1)
popmin	O(log n)	O(log n)	O(log n)
push	O(log n)	O(log n)	O(1)
decreasekey	O(log n)	O(log n)	O(1)
delete	O(log n)	O(log n)	O(log n)
merge	O(n)	O(log n)	O(1)

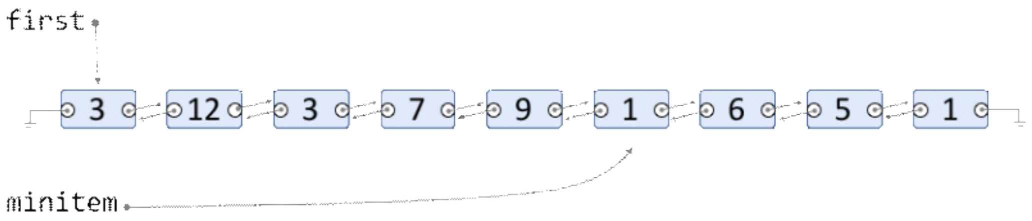
Recall: a **binary heap** is an almost-full binary tree (i.e. every level except possibly the bottom is full), and it satisfies the heap property (each node's key is $\leq$ its children), so it's easy to find the minimum of the entire heap.



Recall: a **binomial heap** is a collection of binomial trees, each satisfying the heap property, and each of different order. A binomial tree of order *k* contains 2^k nodes, has depth $\log_2(k)$, and has a root node with *k* children.



Why not something simpler, a **doubly linked list**? If we really want `push()` and `decreasekey()` to be fast, we could just store all the items in a doubly linked list, and also keep a pointer `minitem` to the item with the smallest key.



`push(v, k):`
just attach the new item to the front of the list, and if `k<minitem.key` then update `minitem`;
this takes *O*(1) time

`decreasekey(v, newk):`
update v's key, and if `newk<minitem.key` then update `minitem`;
this takes *O*(1) time

`popmin():`
we can remove `minitem` in *O*(1) time, but to find the new `minitem` we have to traverse the entire list which takes *O*(*n*) time.

General idea

- The binary heap and binomial heap do cleanup after every `push()` or `decreasekey()`, so that the data structure is kept in a compact shape, and the next `popmin()` can be done quickly.
- The linked list implementation does hardly any cleanup after `push()` and `decreasekey()`, so those operations are fast, but `popmin()` is very slow.

Can we get the best of both worlds?

The general idea behind the Fibonacci heap is that we should be lazy while we're doing `push()` and `decreasekey()`, only doing *O*(1) work, and just accumulating a collection of unsorted items; and we'll only do cleanup (into something resembling a binomial heap) on calls to `popmin()`.

If we're accumulating mess that will have to be cleaned up anyway, why not just cleanup as we go? The heart of the answer lies in our analysis of heapsort in Section 2.12. We saw that it takes time *O*(*n* log *n*) to add *n* items to a binary heap one by one, but only $O(n)$ to heapify them in a batch. Doing cleanup in batches is the big idea behind the Fibonacci heap.

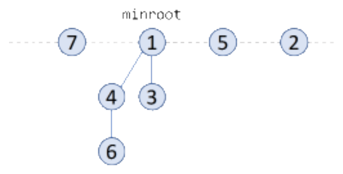
* It turns out that we need creative accounting to properly account for "allow some mess to build up, and cleanup from time to time in batches". So far our complexity analyses have looked at worst-case performance on each individual operation. But with the Fibonacci heap, it's reasonable to say that the cleanup cost should be *amortized* across the operations that made the mess. Complexity analysis will be left to Section 6.3.

Implementation

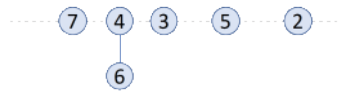
Simple version: without decreasekey()

```
1   # Maintain a list of heaps, i.e. their root elements
2   roots = []
3
4   # Maintain a pointer to the smallest root
5   minroot = null
6
7   def push(Value v, Key k):
8       create a new heap n consisting of a single node (k,v)
9       add n to the list of roots
10      update minroot if n.key < minroot.key
11
12  def popmin():
13      take note of minroot.value and minroot.key
14      delete the minroot node, and promote all its children to be roots
15      # cleanup the roots
16      while there are two roots with the same degree:
17          merge those two roots, by making the smaller root a child of the larger
18      update minroot to point to the root with the smallest key
19      return the value and key we noted in line 13
```

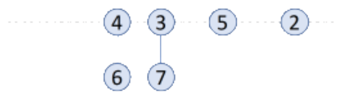
0. typical state



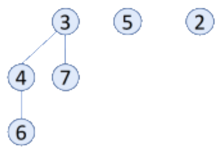
1. `popmin()` deletes `minroot`, and promotes its children to the root list



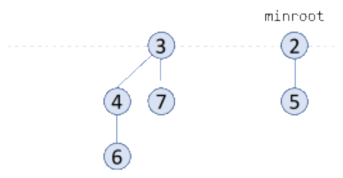
1b. merge two trees of degree 0



1c. merge two trees of degree 1



1d. merge two trees of degree 0, and update `minroot`



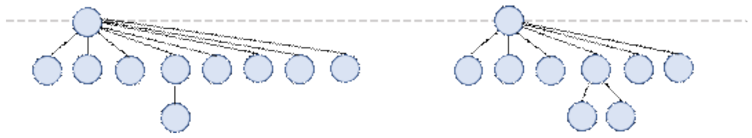
The cleanup rule in lines 16–17 says to merge roots with the same degree, by making one root a child of the other. If both of them are roots of binomial trees of order k, then the merged tree will be a binomial tree of order k+1. The `push()` rule in line

8 will only ever create binomial trees of order 0. Therefore, at all times, we have a collection of binomial trees; and after cleanup we have a binomial heap.

Full version: with decreasekey()

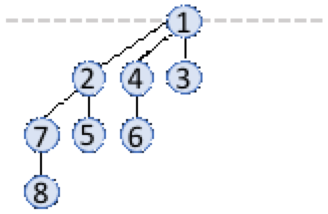
If we can decrease the key of an item in-place (i.e. if its parent is still ≤ the new key), then that's all that `decreasekey()` needs to do. If however the node's new key is smaller than its parent, we need to do something to maintain the heap. We've already discussed why it's a reasonable idea to just cut such a node out of its tree and dump it into the root list, to be cleaned up in the next call to `popmin()`.

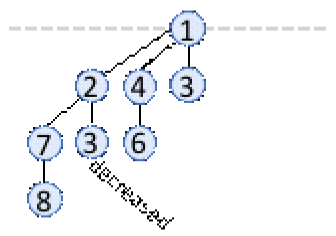
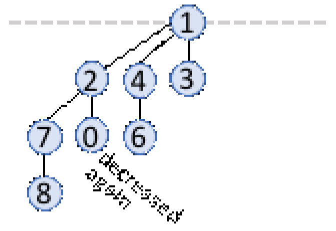
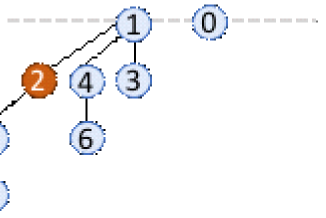
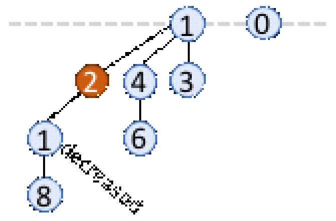
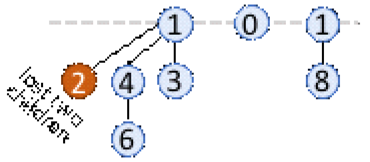
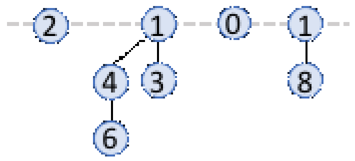
There is however one extra twist. If we just cut out nodes and dump them in the root list, we might end up with trees that are shallow and wide.



To make `popmin()` reasonably fast, we need to keep the maximum degree small. We managed this in a binomial heap: a binomial tree of order k has 2^k nodes, from which one can prove that a binomial heap with n items has maximum degree $O(\log n)$. This suggests that decreasekey() should do some pruning along the way, to make sure that the trees end up with a good number of descendants i.e. without too many childless nodes. They won't be perfect binomial trees, but they won't be too far off.

```
# Every node will store a flag, p.loser = True / False
# Root nodes always have p.loser = False
# Nodes are marked loser=True when they loose one child
# If a loser node p looses its second child, cut out p and make it a root.

def decreasekey(Value v, Key newk):
    let n be the node where this value is stored
    n.key = newk
    if n still satisfies the heap condition: return
    while True:
        p = n.parent
        remove n from p.children
        insert n into the list of roots, updating minroot if necessary
        n.loser = False
        if p.loser = True:
            n = p
        else:
            if p is not a root: p.loser = True
            break

def popmin():
    mark all of minroot's children as loser = False
    then do the same as in the "simple version"
```

0. typical state 		
1. decrease 5 to 3		

		
2. decrease again to 0 	2b. move 0 to maintain heap; 2 is a loser 	
3. decrease 7 to 1 	3b. move 1 to maintain heap 	3c. move the loser 2 to root 

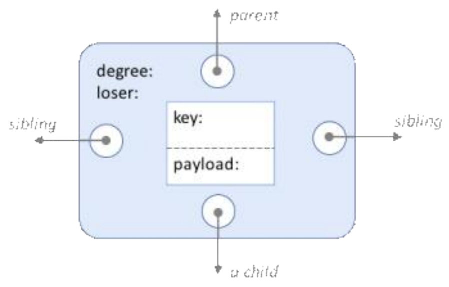
Other operations

- To delete a value *v*, just call `decreasekey(v,-∞)` then `popmin()`. This takes *O*(log *n*) time.
- To merge two Fibonacci heaps, just join their root lists. This takes *O*(1) time.
- To peek at the smallest element without deleting it, just inspect `minitem`. This takes *O*(1) time.

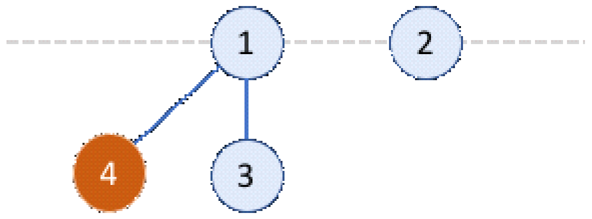
Nuts and bolts of the implementation

All problems in computer science can be solved by adding a layer of indirection; and all of the steps we've described (cutting nodes out of their tree, promoting nodes to the root list, merging trees) can be achieved in *O*(1) time by keeping enough pointers around. We can use a circular doubly-linked list for the root list; and the same for a sibling list; and we'll let each node point to its parent; and each parent will point to one of its children.

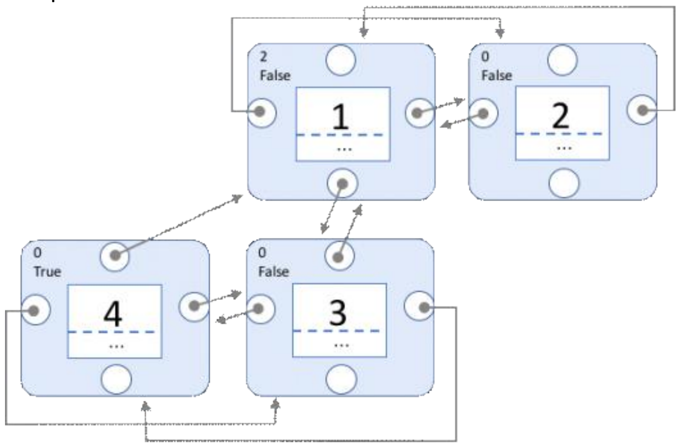
Each node looks like this:



This Fibonacci heap:



is represented as:



6.2 Creative accounting

Aggregate analysis

Consider the "simple version" of the Fibonacci Heap. This supports two operations, `push()` and `popmin()`. What are their running times?

- All `push()` does is update a handful of pointers associated with the new element, so its running time is $O(1)$.
- `popmin()` first promotes minroot's children to be roots themselves, which takes time $O(D)$ where D is the maximum degree among all the roots in the heap. If we started with R roots, we now have D+R. Next, `popmin()` runs a cleanup procedure. Here's a fuller version of how we might implement cleanup:

```
1  newroots = [null, null, ....]   # empty array of length D+1
2  for each r in the list of roots:
3       x = r
4       while newroots[x.degree] is not null:
5           x = merge(x, newroots[x.degree])
6           newroots[x.degree] = null
7       newroots[x.degree] = x
8  roots = list of non-null values in newroots
```

One way to analyse this is to say: the maximum number of iterations of lines 5–6 is D, and the for loop in line 2 has $D + R$ iterations (i.e. the number of roots we started with), so the total running time is $O(D^2 + R\,D)$.

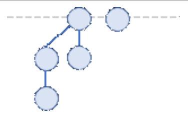
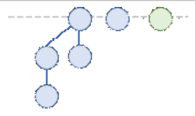
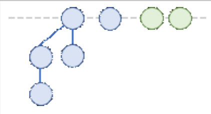
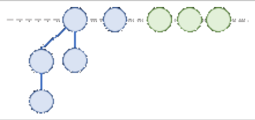
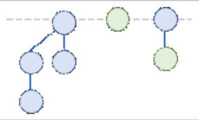
A more sensible analysis is as follows. Every time line 5 is run, we merge two roots. We started with $D + R$ roots, so the maximum number of merges is $D + R$, so the total running time is $O(D + R)$. This style of analysis is called **aggregate analysis**. It just means "Work out the worst-case total cost of a sequence of operations".

We used aggregate analysis throughout our analyses of graph algorithms. For example, we said that depth-first search has running time $O(V + E)$, because it investigates each vertex and each edge at most once; a naive worst-case analysis would have said "At every vertex we look at $O(E)$ neighbours, and there are V vertices in total, so the total running time is $O(V\,E)$."

Accounting method

Some items, like fridges, are expensive to dispose of. If it costs £X for the item and £Y to dispose of, I could treat it as costing £(X+Y): pay £X, put £Y in the bank, and I'll be sure I'll have enough money to dispose of it when the time comes. The same accounting trick can be used for aggregate analysis of running time, as follows:

Suppose we start with a Fibonacci Heap in state $\mathcal{S}_0$, and it has just had cleanup run; and we then do N `push()`es followed by 1 `popmin()`.

initial state $\mathcal{S}_0$ max degree D = 2	push()	push()	push() N = 3 new roots	popmin() $\leq D$+1+N roots cleaned up
				

Each push takes a constant number of elementary operations. If the maximum degree among all roots is D, then $\mathcal{S}_0$ had at most D + 1 roots (we assumed $\mathcal{S}_0$ had just been cleaned up, so no two roots have the same degree), so `popmin()` has up to D + 1 + N roots to clean up, which takes up to $O(D + N)$ elementary operations. The aggregate running time for this full sequence of operations is therefore $O(D + N)$. Let's use the accounting trick, and ascribe the +N part of this cost to each of the N push operations: we'll declare:

- `push()` has amortized cost $O(1)$
- `popmin()` has amortized cost $O(D)$

What do we actually mean by **amortized cost**? We just mean that aggregate true cost is less than or equal to the aggregate amortized costs. In other words, for any sequence of operations consisting of N_1 `push()`es and N_2 `popmin()`s,

$$\begin{aligned}\text{worst-case aggregate true cost} &= N_1\,O(1) + N_2\,O(D)\\ &= O(N_1 + N_2\,D).\end{aligned}$$

Why is this useful? Without this accounting trick, the only precise statement we can make about the running time of `popmin()` is that it is $O(R)$ where R is the number of roots, and $R = O(n)$ where n is the total number of elements in the heap. If we want to bound the running time of a sequence of operations, then the statement "`popmin()` is $O(n)$" does not give us tight answers.

The potential method

Some data structures have a natural way to measure "stored-up mess that has to be cleaned up eventually". Suppose there is a function Φ, called the **potential function**, that maps possible states of the data structure to non-negative real numbers. This gives us a systematic way to apply the accounting trick: let c be the true cost of an operation, let $\mathcal{S}$ be the state of the data structure just before, let $\mathcal{S}'$ be the state just after, and define the **amortized cost** of the operation to be

$$c + \Phi(\mathcal{S}') - \Phi(\mathcal{S})$$

In an aggregate analysis, for a sequence of operations

$$\mathcal{S}_0 \longrightarrow_{c1} \mathcal{S}_1 \longrightarrow_{c2} \mathcal{S}_2 \longrightarrow_{c3} \cdots \longrightarrow_{ck} \mathcal{S}_k$$

where the true costs are $c_1, \ldots, c_k$,

total amortized cost

$$= [-\Phi(\mathcal{S}_0) + c_1 + \Phi(\mathcal{S}_0)] + [-\Phi(\mathcal{S}_1) + c_2 + \Phi(\mathcal{S}_2)] + \cdots + [-\Phi(\mathcal{S}_{k-1}) + c_k + \Phi(\mathcal{S}_k)]$$
$$= c_1 + \cdots + c_k + \Phi(\mathcal{S}_0) - \Phi(\mathcal{S}_k)$$
$$= \text{total true cost} - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k)$$

It's convenient to set $\Phi=0$ when the initially empty data structure is created, and we'll require that $\Phi \geq 0$ for all possible states. This guarantees that the total true cost *of any* sequence of operations is $\leq$ the total amortized cost.

Example

To illustrate, let's look again at the Fibonacci Heap (simple version), and define the potential function

$$\Phi(\mathcal{S}) = \text{number of roots in } \mathcal{S}$$

What are the amortized costs of `push()` and `popmin()`, with respect to this potential function?

- `push()` requires $O(1)$ elementary operations, and it increases Φ by 1, so it has amortized cost $O(1)$.
- `popmin()` first promotes its children to the root list, which takes $O(D)$ elementary operations, and increases Φ by $O(D)$. It then runs cleanup, which involves merging a certain number of trees: every merge takes $O(1)$ elementary operations, and decreases Φ by 1. So cleanup has already been "paid for", and the total amortized cost is just $O(D)$.

We should really be more precise when we say "$O(1)$ elementary operations are cancelled out by $\Delta\Phi = -1$". By $O(1)$ we just mean "a constant number of elementary operations", and we'll take 1 unit of Φ to be worth the largest of these constants.

The potential method has given us exactly the same amortized costs that we proposed for the accounting method.

6.3 Analysis of Fibonacci heap

Let's derive the amortized costs in the "full version" Fibonacci Heap, using the potential function

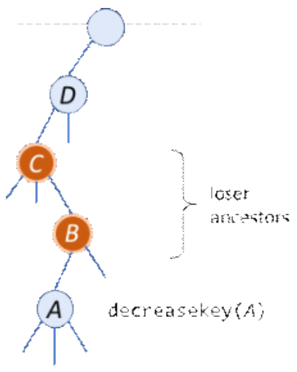
$\Phi(\mathcal{S})$ = number of roots in $\mathcal{S}$ + 2 (number of loser nodes in $\mathcal{S}$)

Amortized cost of operations

`push()` takes $O(1)$ elementary operations, and increases Φ by 1, so the amortized cost is $O(1)$.

`popmin()` first promotes its children to the root list and marks them as not losers, which takes $O(D)$ elementary operations and increases Φ by $O(D)$. It then runs cleanup, which involves merging some number of trees t, which takes $O(t)$ work and also decreases Φ by t. So the amortized cost is $O(D)$.

`decreasekey()` takes $O(1)$ elementary operations to decrease the key. If the node doesn't have to move, then Φ doesn't change, so amortized cost = true cost = $O(1)$.



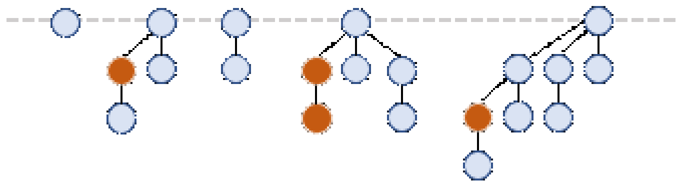
If the node A does have to move,

1. Move A to the root list. True cost is $O(1)$, and Φ increases by ≤ 1 (it increases by 1 if A wasn't a loser, and decreases by 1 if it was.)
2. Move some loser ancestors B, C to the root list. For each of these, the true cost is $O(1)$, and Φ increases by 1 because there's a new root, and decreases by 2 because the node gets marked as not a loser. Thus the amortized cost of this part is zero, regardless of the number of loser ancestors.
3. One ancestor D might have to be marked as a loser. True cost is $O(1)$, and Φ increases by 2.

The total amortized cost is $O(1)$. We can see now why the potential function had to be defined the way it was: so that the amortized cost of step 2 does not depend on the number of loser ancestors.

Bounding the shape of the trees

The amortized cost of `popmin()` is $O(D)$, where D is the maximum number of children in any of the trees. The peculiar mechanism of `decreasekey()` was designed in order to keep D small. How small? The answer rests on a theorem, which we will now prove (and which should remind you of Exercise 37).



Theorem. In the Fibonacci heap, if a node has d children, then the total number of nodes in the subtree rooted at that node is at least F_{d+2}, the $(d+2)$nd Fibonacci number.

The exact value of the Fibonacci numbers doesn't matter. What matters is the following corollary, which is easy to prove, given the mathematical fact that F_d grows exponentially: $F_{d+2} \geq \varphi^d$, where $\varphi = (1+\sqrt{5})/2$ is the golden ratio. (See your answer to Exercise 50, about the maximum degree in a binomial heap, for a similar proof.)

Corollary. In a Fibonacci heap with n items, $D = O(\log n)$.

Proof of theorem.
First, some observations about the execution of the algorithm. Take any node x in a Fibonacci heap, and suppose it has d children. Consider the evolution of the heap, up to the point in time T that x has these children. They were made x's children in some order, say y_1 then ... then y_d. When y_k was made a child of x, at time T_k say, x had $\geq k$-1 children: $y_1 \ldots y_{k-1}$, and possibly some others that have since been cut out. By the rule for cleanup, lines 16–17 in `popmin()`, y_k had the same number of children as x at time T_k, namely $\geq k$-1. Now, at time T, y_k must have $\geq k$-2 children: any fewer, and it would have been cut out of the tree by lines 30-–35 of `decreasekey()`.

Next, some mathematical observations. Let N_d be the minimum possible number of nodes in a subtree whose root has degree d. Using what we have just observed about the number of x's grandchildren,

$N_0 = 1$ (a node with no children)

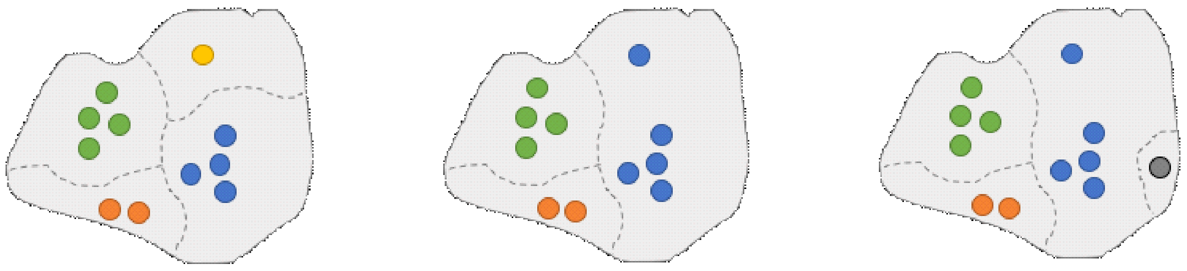
$N_1 = 2$ (a node with one child and no grandchildren)

$N_d = 1 + 1 + N_0 + \cdots + N_{d-2}$ (the root node + y_1's subtree + y_2's subtree + $\cdots$ + y_d's subtree)

$= N_{d-1} + N_{d-2}$ (by substituting in the expression for N_{d-1})

and these are the defining equation for the Fibonacci numbers, only offset by 2.
QED.

6.4 Disjoint sets



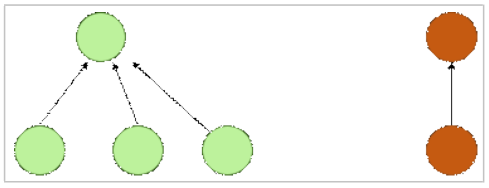
The DisjointSet data structure (also known as union-find or merge-find) is used to keep track of a dynamic collection of disjoint sets. We used it in Kruskal's algorithm for finding a minimum spanning tree: each individual item was a vertex of a graph, and we used sets to track which vertices we had joined together into tree fragments.

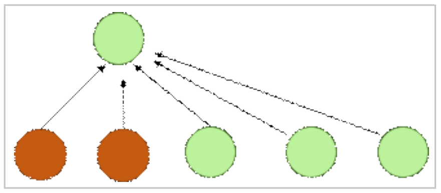
Abstract data type DisjointSet

```
ADT DisjointSet:
  # Holds a dynamic collection of disjoint sets

  # Return a handle to the set containing an item.
  # The handle must be stable, as long as the DisjointSet is not modified.
  Handle get_set_with(Item x)

  # Add a new set consisting of a single item (assuming it's not been added already)
  add_singleton(Item x)

  # Merge two sets into one
  merge(Handle x, Handle y)
```

The specification doesn't say what a handle *is*, only that the handles don't change unless the DisjointSet is modified (by either `add_singleton()` or `merge()`). In practice, we might use a representative element from each set as the set's handle.

Implementation 1: flat forest

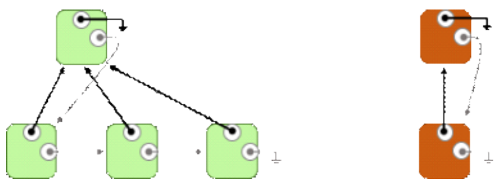




To make `get_set_with()` fast, we could make each item point to its set's handle.

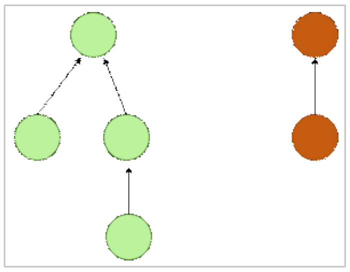
- `get_set_with()` is just a single lookup.
- `merge()` needs to iterate through all the items in one or other set, and update its pointer. This takes *O*(*n*) time, where *n* is the number of items in the DisjointSet.

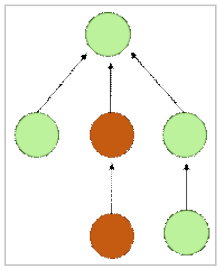
To be able iterate through the items, we could store each set as a linked list:



A smarter way to `merge()` is to keep track of the size of each set, and pick the smaller set to update. This is the **weighted union** heuristic. It can be shown that the aggregate cost of any sequence of *m* operations on *n* elements (i.e. *m* operations, of which *n* are `add_singleton()`) is *O*(*m* + *n* log *n*).

Implementation 2: deep forest





To make `merge()` faster, we could skip all the work of updating the items in a set, and just build a deeper tree.

- `merge()` attaches one root to the other, which only requires updating a single pointer
- `get_set_with()` needs to walk up the tree to find the root. This takes $O(h)$ time, where h is the height of the tree.

To keep h small, we can use the same idea as with the flat forest: keep track of the **rank** of each root (i.e. the height of its tree), and always attach the lower-rank root to the higher-rank. If the two roots had ranks r_1 and r_2 then the resulting rank is $\max(r_1,r_2)$ if $r_1 \neq r_2$, and r_1+1 if $r_1=r_2$. Call this the **union by rank** heuristic.

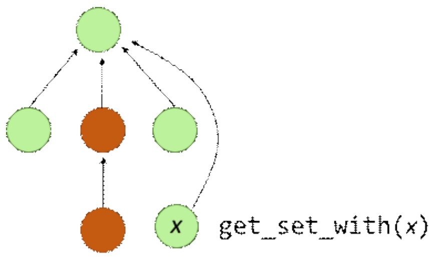
Implementation 3: lazy forest

We'd like the forest to be flat, so that `get_set_with()` is fast; but we'd like to let the forest get deep so that `merge()` can be fast.

A smart idea is to flatten the forest lazily:

- Implement `merge()` as in the deep forest.
- Whenever `get_set_with(x)` is called, walk up the tree to find the root; and then walk up the tree a second time and make x and all the intermediate nodes be direct children of the root.

This is called the **path compression** heuristic. We won't adjust the stored ranks during path compression (and so rank won't be the exact height of the tree, just an upper bound on the height.)



It can be shown that with the lazy forest the cost of m operations on n items is $O(m\,\alpha_n)$, where α_n is an integer-valued monotonically increasing sequence (related to the Ackerman function) that grows extremely slowly:

$\alpha_n = 0$	for n=0,1,2
$\alpha_n = 1$	for n=3
$\alpha_n = 2$	for n=4,5,6,7
$\alpha_n = 3$	for $8 \leq n \leq 2047$
$\alpha_n = 4$	for $2048 \leq n \leq 10^{80}$, which is more than there are atoms in the observable universe

For practical purposes, α_n may be ignored in the O notation, and therefore the amortized cost per operation is $O(1)$.